ADA AS AN IMPLEMENTATION LANGUAGE FOR KNOWLEDGE BASED SYSTEMS

Daniel Rochowiak
Research Scientist

Johnson Research Center
University of Alabama in Huntsville
Huntsville, AL 35899

ABSTRACT

Debates about the selection of programming languages often produce cultural collisions that are not easily resolved. This is especially true in the case of Ada and knowledge based programming. The construction of programming tools provides a desirable alternative for resolving the conflict.

INTRODUCTION

If one wants to generate a debate at a party for persons connected with computer programming, just ask "What is the best programming language?" The result is often an outpouring of praise, curses, hyperbole, and technical detail that will either quicken the pulse or induce tranquil repose. Programming languages are at times treated as matters of religious fervor, and at other times treated as mere notational convention. All of this would be fine were it not for the demands for "good" software and the increasing size, complexity and seriousness of software programming projects. To be sure software is more than the code for a program. Software, in the sense includes all of the information that is: (1) structured with logical and functional properties, (2) created and maintained in various representations during its life-cycle, and (3) tailored for machine processing. This information in large projects is often used, developed, and maintained by many different persons who may not overlap in their roles as users, developers, and maintainers. In order to develop good software, one must explicitly determine user needs and constraints, design the software in light of these and in light of the needs and constraints of the implementers and maintainers, implement and test the source code, and provide supporting documentation. These dimensions and constraints on producing software can be looked at as aspects of different moments in the software production process.

The programming languages LISP and Ada can each legitimately claim a special competence. In the case of LISP, it is symbolic processing, and in Ada, uniformity and maintainability. In making a decision about a programming language, the programming language and its environment cannot be meaningfully separated. Whether one examines LISP or Ada, it is clear that the advocates of these languages are not considering the languages in isolation. The combination of programming environment and programming language is intimately connected with the programming paradigm that can be used in the construction of the program. A programming paradigm may be thought of as the style or manner in which a program is created. Within one paradigm there may be many particular templates, but there is a sense in which each of these reduces back to some primitive template. Alternatively, one may view the paradigm as a primitive object from which the specific template inherits structures and properties. Under either sort of interpretation, it should be clear that a programming paradigm acts as a vehicle through which a programmer designs and builds specific programs.

Certainly another way to generate debate is to ask, "What is the best representation of knowledge?" or "What is the best way to manipulate knowledge?" The list of answers will grow rapidly: logic, rules, frames, scripts, objects, trees, nets, inferences, associations, statistical inferencing, case based reasoning, analogy, and so on. All of these styles and techniques have valued uses. All have their strengths and weaknesses. Unless a person was very lucky, no consensus would be achieved at the party.

Behind both the questions about programming languages and the questions about knowledge is a common social structure. Programming and the construction of knowledge based systems occur in cultures. These cultures are the repository for tradition, tacit rules of procedure, and tacit rules of appraisal. A person's training is the way in which they are enculturated. Someone who is trained on a certain hardware, in a certain language, and in a certain style will carry the culture generated through that training onto his or her new works. As persons with their cultures collide differences of opinion, and difficulties in adjusting to the demands of another are sure to be produced. This collision of cultures is a central element of the issues surrounding the debates about knowledge based programming and Ada.

PARADIGMS AND CULTURES

Typically a culture has a core paradigm or set of paradigms that capture the core of the culture. The paradigms act as cognitive templates that are filled in when either trying to solve a problem or develop an object.

In programming there is an interaction between what may be considered a programming paradigm and a programming language. Stroustrup (8) sets out the relation between a programming paradigm and a programming language rather neatly.

> A language is said to support a style of programming if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style.... Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in subtle forms of compile-time and/or run-time checks against unintended deviations from the paradigm... Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for a paradigm.

The problem of selecting a paradigm is both art and science. It is art insofar as it requires a subtle understanding of the programming craft, and is science insofar as a set of decision rules can be established for the paradigm. The four typical paradigms are: procedural, data hiding, abstract data type, and the object-oriented Paradigm.

In examining Ada and LISP the idea of the programming paradigm can be usefully extended to the paradigmatic way in which programming languages and their environments are used. The question is whether one language offers tools that make it best suited to a particular task. Although there is a formal sense in which all sufficiently rich languages are equivalent, this equivalence is only logical or formal. Although any of the paradigms can be accomplished in either LISP or Ada , this does not mean that it is either easy or reasonable to use any of the mixtures of paradigm and language that are possible. Since LISP has been the chief language of artificial intelligence research, it is reasonable to investigate whether Ada can support the constructs of LISP. In this way the issue concerns whether Ada can implement the LISP paradigm.

Schwartz and Melliar-Smith (7) analyzed the Ada specification to determine its potential as an AI research language. Their conclusion is that Ada, as defined in the Preliminary Standard,

would not be suitable as a "mainstream research language." They proposed, however, that with some extensions it is plausible that a substantial portion of AI "algorithms" could be translated into Ada. This translation would not be easy, since it would be more of a "reimplementation" of the program, but the "complex heuristic algorithms that provide the artificial intelligence" could be retained.

Schwartz and Melliar-Smith's claim of Ada's unsuitability is fundamentally based on the determination to enforce a particular programming paradigm. One goal that was set forth in both the Ironman and Steelman Requirements, is to create "an environment encouraging good programming practices." Ada imposes a style of programming that is the result of many years of research on programming methodology. Ada is intended to impose a very disciplined style of programming that assists those who are developing large, complex projects that require teams of programmers. Furthermore, Ada is said to be 'readable and understandable rather than writeable' so as to minimize the cost of program maintenance. Thus, Ada's mandated programming style is beneficial for the targeted Ada community - a production community, especially a community that produces real-time embedded systems. In general, AI work does not occur in a production community, but a research and development community. This difference in orientation is a factor in making Ada unsuitable as a general AI research language. The constraints of production prevent the AI programmer from using the most natural method of expression for whatever system is being developed. The LISP programmer places greater value on code that is more easily writeable than readable. However, two things should be remembered. First, the readability of any code is a function of the enculturation of the reader. Second, the readability of the code is a function of the tools available with which to read it. This latter point is important when one considers LISP on a LISP machine. Within that environment the code may become very readable through the tools that are available for reading it.

Schwartz and Melliar-Smith contend that the utility of Ada for AI programs is confined to the reimplementation. This operation would be carried out by software teams by following the algorithms of an original program, but not necessarily its detailed code. Extensions to Ada are needed, however, if such reimplementation is to be carried out while preserving Ada's structure and modularity.

A typical AI task for a knowledge based system in LISP is to generate solutions to problems that have a very large number of alternatives. To attempt to solve such a problem by exhaustive search or "best fit" is not feasible even with a supercomputer. A heuristic based guess is used to prune branches from the decision tree so that the problem becomes tractable. In some "classic" systems, a breadth-first or depth-first search is used to consider candidate solutions. When it becomes apparent that an incorrect decision has been made, then the search resumes at the junction where that decision was made. Use of heuristics allows for systems to "learn" from their mistakes and refine their search techniques as more is "learned" about the problem domain. Several features of AI programs stand out. First, extensive use is made of the list structure and the processing of lists. Second, procedures are often used as values that can be stored in a data structure. This allows for the construction of a generic framework for the parameterized transformation of a given type of structure. An example of this would be the construction of a system to perform an arbitrary function on a tree or graph structure. If the procedures are values of a procedural data type, then the procedures could be passed as parameters to perform the desired manipulation of data. Third, LISP provides for a similar representation of both data and programs that allows for the creation of functional abstractions "on the fly." These abstractions, expressed by Lambda calculus list expressions, can be passed as parameters to other abstractions. Fourth, as each function or expression is defined, it becomes part of the system. Thus, the application program can examine its run-time environment, a fact which makes the program inseparable from its environment. Finally, the ability to use procedures as storable objects is essential to many AI programs. One use for this ability is as a method to express knowledge about a particular domain. Frequently, several different knowledge representations will be used

in one system. The particular representation used would depend on the availability of information.

PACKAGES FOR ADA

Much of the success of LISP as an AI language can be attributed to fact that it is extensible. It is possible, for instance, to construct rather easily interpreters for other high-level languages using LISP. This ability is facilitated by the manner in which LISP programs are represented: as lists. Ada, too is extensible. (2) However, Ada is more limited in its extension capabilities, with packages, generic procedures and tasks being all of the extension methods. Whether or not this extensibility is to limited for the needs of reimplementing AI programs remains to be seen. Ada provides data abstraction facilities that allow one to create extensions to the language by the defining of new data types and the operators that can be used to manipulate them. Through the use of a package containing a data abstraction, a programmer can write code as if the facilities provided by the package were provided by Ada. Thus the addition of packages may provide a way in which the typical features of an AI program written in LISP can be reimplemented in Ada. Such a package may, for example, supply the tools needed to handle lists, procedures, and garbage collection.

List processing is an important feature of AI research languages. Whereas Ada does provide the features needed to implement list processing, its garbage collection facilities leave much to be desired. No special considerations have been made for list processing, and consequently, the efficiency of such will likely be minimal. To implement lists in Ada, one could create a data structure as follows. Each list cell would be a record that has two list pointers: CAR and CDR. A list pointer would then be a record that has only a variant part. The discriminant of this variant part would have two possible values: ATOM and LIST. This would indicate whether the list pointer component is a list reference or an atom reference. There would need to be a LIST_REFERENCE and ATOM_REFERENCE access types for the dynamically allocated list cells and atom cells.

Although procedural variables cannot be readily added to Ada, it is conceivable that the ability to pass procedures as parameters could be added. The effect of the instruction part of a procedural value can be simulated through the use of a generic procedure. This method would avoid using a CASE statement as would be necessary if the indexing scheme were used. Generic procedures used in this fashion would carry the name of the "passed" procedure but would not have the closure or environment.

As most AI programs "run," they pursue a number of possible alternative paths of action. This attempt to find the best possible path usually succeeds in allocating a great deal of memory. Since the memory objects have a lifetime that is dependent on the duration of the utility of the data, and not the flow of control of the program, these objects must be allocated in a global heap. By using a heap, storage can remain at least as long as it is referenced anywhere else in the system. Consider a typical embedded system application. Here, the data that must have space in the heap is minimal. Thus, reclamation of heap space is not important, and in some cases, heap space is not reclaimed at all. This is yet another design philosophy contradiction, between Ada and AI languages. AI languages are designed with the philosophy that "no amount of initial heap allocation will be sufficient for the continued operation of many AI programs." It is not a question of if all of the heap space will become allocated, rather is is a question of when it will happen. Obviously, some strategy must be used to reclaim this storage space. The language specification for Ada does not preclude garbage collection capabilities, nor does it indicate these will be included. There is a mechanism, FOR-USE, which indicates the maximum number of objects of an access type that may be generated. Since the compiler knows the maximum size in advance, the necessary space can be allocated. This provides a sort of heap-type allocation with

automatic reclamation for objects that have a limited scope of use. Unfortunately, this method causes allocation/deallocation to be dependent on control flow or block entry and exit.

PATTERNS

Obviously, not everyone agrees with Schwartz and Melliar-Smith on Ada's place in AI. Larry Reeker, John Kreuter, and Kenneth Wauchope of Tulane University have done much work on pattern matching in Ada. In answer to the question of "will Artificial Intelligence be done in Ada?" they answer that "anything can be done in Ada," and attempt to show how Ada, when appropriately used, can facilitate the programming of Artificial Intelligence applications. (6)

Reeker has chosen to focus on a pattern-directed because "pattern-directed facilities provide the most effective means for creating complex programs for non-numerical applications." Further support for pattern matching can be found in the work of Warren, Pereira and Pereira. (9) They contend that pattern matching "is the preferable method for specifying operations on structured data, both from the user's and the implementer's point of view."

Reeker envisions the addition of AI oriented features to Ada through the use of packages. The list of features that are candidates for incorporation into Ada include:

- String definition and manipulation facilities more flexible than those built into Ada.
- List processing functions
- Pattern definition and matching functions for strings and lists
- A means of manipulating lists returned by the pattern matching functions

Ada's concurrency paradigms lead to a number of possible methods for pattern matching. One such method would be to use tasks as "coroutines" to match patterns. There are areas in AI which have made use of "quasi-parallel" processes previously. True parallel tasks executing on a true multiprocessor system would surely improve on those systems.

In his section of their paper, Kenneth Wauchope presents an Ada language implementation of a pattern-directed list processing facility. A set of SNOBOL-4 like primitives are used to construct lists that are equivalent to arbitrarily complex LISP-like data structures. Wauchope advocates the addition of packages to make AI feasible in Ada. In this paper he describes the operation of a package which provides basic list creation and manipulation functions similar to those in LISP. Wauchope then presents several applications of these new features, including: parsing a context free grammar and symbolic differentiation.

Krueter presents several algorithms for pattern matching in Ada The first of these is the recursive descent parsing method which is a common way to implement the backtracking strategy. Backtracking is based on the intuitive approach of trying every possibility for each pattern element. This generates every possible parse of the string but is rather costly in terms of time.

One particularly interesting possibility arises with the use of Ada for coding such algorithms. Since Ada allows concurrent tasks, the backtracking aspects of the algorithm could be achieved through the use of tasks that behave as coroutines. A task would start by examining each bead in the first set of alternatives. A new task is forked for each successful match. This new tasks will then examine the remainder of the string and the remaining sets of alternatives. After all alternatives have been examined, the task will pass back the matching substrings, or null in the case of no match, and terminate. Each successive parent tasks will then add its substring to the beginning of each tree on the list which has been passed to it. Then, this list is passed back, and so forth, until the master task is reached.

Combinatorially implosive algorithms (CIA's) are a class of parallel algorithms that employ two or more algorithms running concurrently such that they will solve a problem more quickly than one would by itself. Brintsenhoff, Christensen, Mangan, and Greco demonstrate a CIA coded in Ada in their paper, "The Use of Ada Concurrent Processing Features in an Implementation of Parallel Tree Searching Algorithms." (3) This study is interesting because the authors had access to a multiprocessor with run-time support for concurrent tasking. Their findings show the speed advantages of parallel algorithms written in Ada. Although the results were highly data dependent, the running of two algorithms concurrently proved to be more efficient than just one and thus proved the utility of CIA's in Ada. If such CIA's could be developed for pattern matching, it is reasonable to expect that pattern driven AI applications would prove to be very efficient in Ada.

OPTIONS

The two previous sections have indicated two ways in which the confrontation of Ada and AI might proceed. In the first way the differences of the two cultures are acknowledged and an effort is made through the addition of appropriate packages to provide the tools for a reimplementation of an AI program. The second option acknowledges the fact that in a sufficiently complete language it is possible to implement the idea of a program directly. Each approach has certain advantages and disadvantages. In the first approach the program does not have to be completely rethought and redesigned. This is a disadvantage of the second option since the ideas for the program have to implemented from scratch. In the second approach there are potential advantages to be gained by using the strengths of Ada. This is the disadvantage of the first option. The addition of the packages may in effect provide for a LISP interpreter that circumvents the natural strengths of Ada.

One way in which a decision between these options might be facilitated is by using the resources of software engineering. Fairley (4), for example, defines software engineering as the "technological discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates," and claims that software engineering is a "new technological discipline distinct from, but based on the foundations of, computer science, management science, economics, communication skills, and the engineering approach to problem solving." Boehm (1) identifies seven basic principles in software engineering. These are:

1. Manage using a phased life-cycle plan,
2. Perform continuous validation,
3. Maintain disciplined product control,
4. Use modern programming practices,
5. Maintain clear accountability for results,
6. Use better and fewer people,
7. Maintain a commitment to improve the process.

Of these principles one requires special attention in this context, injunction to use modern programming practices.

Programming paradigms are at the root of modern programming practices. As Boehm (1) notes, "The use of modern programming practices (MPP), including top-down structured programming (TDSP) and other practices such as information hiding, helps to get a good deal more visibility into the software development process, contributes greatly to getting errors out early, produces understandable and maintainable code, and makes many other software jobs easier, like integration and testing." At issue, of course, is what counts as a modern programming practice. Interpreting this principle is complicated by the facts that modern programming practices are not fixed, that such practices are the outgrowths of programming paradigms, and

that the paradigms are responses to the practical needs of computer software developers and the intellectual demands of computer scientists.

Thus, Boehm's principle that modern programming practices ought to be used is a bit odd. What it might really mean, however, is not that any modern programming practices should be used, but that the modern programming practices for imperative, conventional languages that are used for large software projects and can be handled within the current discipline of software engineering should be used. In this sense the modern programming practices are those geared to the community and culture of production. Neither LISP nor object-oriented programming can satisfy those demands. However, Ada comes near to being the ideal language from the point of view of software engineering with conventional languages. This points out the difficulty in generating a set of principles to guide software engineering. The analogy of software engineering to the rest of the engineering field (10) begins to break as one attends to the nonphysical character of software. For example, when building a bridge or a pipeline, the standard elements of the construction remain static. Bridges will have beams and pipelines will have pipes. The materials and techniques may change, but the basic elements remain. Unconfined by such physical characteristics, the elements of software construction can change. subroutines, subprograms, libraries, modules, package, units, function, objects and many other elements are available to the software programmer, and new as yet unthought of constructs might be added. All of this adds to the complexity of choosing and using modern programming practices, and points to the important role of the software manager even within the software engineering discipline.

The decision as two which of the two options should be pursued any not therefore be decidable on the grounds of software engineering alone. If the other principles that Boehm isolates are essentially management principles then it is fair to assume that they can be satisfied with any language and any programming paradigm. In this sense they are transcultural. However, the injunction to use modern programming practices is what the collision of cultures is about. What is a modern programming practice? Each culture will defend itself as being the exemplar of modern programming practice. Given the existence of the colliding cultures, it does not appear that the principles of software engineering will be able to generate a clear choice.

Another way in which the choice might be made is to focus on a technological solution. In particular the development of software tools that allow for program development in a neutral environment, but can generate code in a target language. The use of automated tools to manage the software coding process, including the generation of source code in a target language, raises another interesting issue, however. If the tools are good tools and if the code they generate is good code, then what is the programmer doing? In a primary sense he or she is running the tool; in a secondary sense he or she is programming in some language. There is a sense in which if the tools are very well done, the "programmer" need not even know the language in which he or she is programming, and, indeed, need not even know in what language the code is being generated. As Howden (5) has noted:

> The manufacture of software is perhaps one of the most logically complicated tasks. The intellectual depth of software systems development coupled with the lack of physical restrictions imposed by properties of the product make software manufacturing intriguing in its possibilities for highly automated, sophisticated manufacturing environments. Research has begun, on environments containing their own concept models of general programming knowledge... It has been speculated that in the future software engineers will be able to describe application programs to a system capable of automatically generating specifications and code.

An intriguing possibility! The programmer is no longer a crafter of code, but an expert user of a tool. The connection between the programming language and the programmer is, in a sense, severed. The programmer with such tools may, therefore, function at a higher, more natural level of abstraction without needing to attend to the syntactic complexities of the language in which the application is finally coded. This is not, however, surprising. It represents simply another moment in the evolution toward higher level languages. Rather than a traditional higher level language being used with a compiler to generate the low level instructions to the processor, a new generation of tools may operate at an even higher level and a translator may then convert the tool's specifications into a higher level language which in turn may be compiled.

If this technological path is found desirable, then it suggests that the first option, the addition of packages and reimplementation of code, is on the right track. Further it suggests that the second options benefits might be incorporated into the tool. If for example the target hardware is a multiprocessor system, then a tool should be able to guide the tool user in creating code appropriate to the hardware. The creation of such tools is not, however, to thought of as revolutionary. Rather the emergence of such is another evolutionary step in generating higher level software facilities for programming more complicated hardware.

SPECULATION

It is clear, for example, that processors have improved greatly over the past two decades. Increased speed, increased word size, augmented capabilities, decreased power consumption, and decreased cost are readily apparent. All of these factors combine to allow those who design and build languages and environments to implement more easily and effectively ideas and constructs which with less capable processors would remain dream and desire. One need only to recall what it was like to run LISP on a PDP-11 under RSTS and look at a Symbolics or Texas Instruments LISP machine to recognize the difference. Similarly, C in its own UNIX environment has come to be recognized as a powerful system, and has led to the evolution and development of the computer workstation. The future holds even more promise. Even as physical limitations begin to affect the development of better processors, new architectures begin to evolve. Multiprocessor machines, parallel processor machines, and other objects of wonder and splendor open new vistas to the language crafter. Although languages like LISP and C will probably move into these new environments, their form and function will probably be much different. Equally probable is that new languages will emerge. In any case, one point is clear. Languages are not static. Language development responds to the state of the processor art. As long as processor development continues, it is reasonable to expect programming languages to develop.

It should also be clear that programming paradigms change over time. The changes of paradigm reflect both the intellectual development of computer programming and the ability of the language crafters to build support for a paradigm into a language. BASIC was a wonderful language. It was criticized for not supporting a structured programming paradigm. New BASIC arose in response to that criticism. Classic LISP did not support object-oriented programming. LISP with FLAVORS is a virtually seamless environment in which such programming is supported and encouraged. C did not support the object-oriented paradigm; C++ is a response as is Objective C. If it were not for the government's involvement with Ada, one might well think that OO-Ada (Object-Oriented Ada) might soon appear. There is, of course, no reason to think that the story ends with the object-oriented paradigm. New paradigms, perhaps tailored to particular classes of problems, may well arise. As they do, old languages may evolve to support them, and new languages may arise to enforce them in much the way that PASCAL and MODULA-2 enforce structured programming, SMALLTALK enforces object-oriented programming, and Ada enforces some software engineering practices.

Perhaps the most dramatic changes of language will occur with the improvement and development of programming environments and tools. It is often the environment that captures

the programmer. The facilities of the LISP and C environments allow the programmer to concentrate on the task at hand, and quickly and efficiently produce the needed code. This is especially true of a LISP environment on a LISP machine. The programmer can build his own tools and tailor the environment to his or her needs and preferences. More importantly, the environment and machine function in harmony to allow the programmer to build new languages in which problems can be solved. By allowing a measure of abstraction, generality and efficiency can be gained. All of these things taken together point out that the developmental environment is an important factor in selecting a languages.

As programming tools and aids evolve, the direct contact with the programming language may begin to disappear. Such tools may allow the programmer to either break the programming task down into parts that are sufficiently small and standard that existing libraries of routines can be employed, or may allow the programmer to build the program specifications in such a way that a translator will be able to translate the specification into the target language. Both approaches currently have their problems. In the former the programmer is left at some point to grapple with the language itself, and in the latter the programmer might find the translated code for the target language indecipherable. Although these are serious problems, they may not be insurmountable. If they can be overcome, the contact of the programmer with the programming language will be stretched thinner and thinner.

The continued improvement of programming tools and environments, may lead the manager to base his or her decision on which programming language to use on the presence or absence of certain features in the tools and environments more than on the characteristics of the languages. The decision, of course, is still affected by external factors. Ada will be used on the Space Station. However, much might be learned by examining the environments and tools for other languages such as LISP and C in an effort to build better tools for Ada. After all, given that Ada is a sufficiently universal language, it can be made to look like other languages.

CONCLUSION

It is difficult if not impossible to directly solve the cultural collisions that are bound to occur in the interaction of programming languages, and paradigms. Those cultural collisions will not be resolved by attempting to enforce a uniform programming language and culture. An alternative, however, is to build tools that remove the programmer from direct contact with the programming language. This removal can allow the tool user to overcome the cultural problems, while still allowing the production of code in a desired language. If and when such tools become available, the questions with which this essay started will be displaced with the question, "What can your tool do?"

ACKNOWLEDGEMENTS

This research has been partially funded under NAS8-36955 (Marshall Space Flight Center) D.O. 34 "Applications of Artificial Intelligence to Space Station."

REFERENCES

1. BOEHM, B.W. "Seven Basic Principles of Software Engineering," *The Journal of Systems and Software* 3, (1983), pp. 3-24.

2. BOOCH, G. *Software Components with Ada: Structures, Tools, and Subsystems*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1987.

3. Brintsenhoff, Alton; Christensen, Greco, Joe; Steve; Mangan, John. "The Use of Ada Concurrent Processing Features in an Implementation of Parallel Tree Searching Algorithms. *Proceedings of the Third Annual Conference on Artificial Intelligence and Ada*. George Mason University, October, 1987.

4. FAIRLEY, R.E. *Software Engineering Concepts*. McGraw-Hill Book Company, New York, 1985.

5. HOWDEN, W.E. "Contemporary Software Development Environments," *Communications of the ACM*, 25, 5 (1982), pp. 318-329.

6. REEKER, LARRY H.; KREUTER, JOHN; WAUCHOPE, KENNETH. "Artificial Intelligence: Pattern-Directed Processing." Final Report AFHRL-TR-85-12 Air Force Human Resources Laboratory, Lowry Air Force Base, Colorado. May 1985.

7. SCHWARTZ, RICHARD L AND MELLIAR-SMITH, P.M. "On the Suitability of Ada for Artificial Intelligence Applications." Final Report for Defence Advanced Research Projects Agency Contract DAAG29-79-C-0216. July 1980.

8. STROUSTRUP, B. "What is 'Object-Oriented Programming'?," *ECOOP '87: European Conference on Object-Oriented Programming, Paris, France, June 15-17, 1987, Proceedings*. [Bézivin, J., P. Cointe, J.-M. Hullot, and H. Lieberman (Eds.)]. *Lecture Notes in Computer Science (276)*, Goos, G. and J. Hartmanis (Eds.), Springer-Verlag, Berlin, 1987.

9. WARREN, D. H. D.; PEREIRA, L. M.; PEREIRA, F. "PROLOG - the language and its implementation compared with LISP." *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, Rochester, New York, pp 109-115. 1977.

10. ZELKOWITZ, M.V., A.C. SHAW, and J.D. GANNON. *Principles of Software Engineering and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1979.